Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Czwartek 19:05*
Temat *Algorytmy lokalnego przeszukiwania*	**Problem** *TSP*
Skład grupy *241281 Karol Kulawiec*	**Nr grupy** -
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *9 kwietnia 2020*

1 Opis problemu

Podczas pierwszego etapu należało napisać algorytmy lokalnego przeszukiwania rozwiązujące problem komiwojażera. Problem komiwojażera polega na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Jest to problem NP-trudny, co oznacza, że nie jest możliwe rozwiązanie problemu komiwojażera ze złożonością obliczeniową wielomianową.
Zbadane algorytmy lokalnego przeszukiwania: Simulated Annealing oraz Tabu Search.

2 Metoda rozwiązania

2.1 Simulated Annealing

Algorytm Simulated Annealing (Symulowane wyżarzanie) to algorytm heurystyczny, przeszukujący przestrzeń alternatywnych rozwiązań, które ze sobą sąsiadują, dobranych w sposób losowy, w celu znalezienia rozwiązań najlepszych. Nawiązuje do zjawiska wyżarzania w metalurgii, w którym próbka metalu poddawana jest procesowi studzenia w celu osiągnięcia pożądanych cech. Algorytm spośród swoich sąsiadów wybiera najlepsze rozwiązanie i udaje się w tamtym kierunku, jeżeli nie znajdzie rozwiązania lepszego, to z pewnym prawdopodobieństwem, zmniejszającym się wraz z temperaturą, przyjmie gorsze rozwiązanie, dzięki czemu nie utknie w lokalnym ekstremum.

Rozwiązania losowe są generowane przez odpowiednio 3 funkcje: swap(a,b), insert(a,b) i invert(a,b). Dla każdego z nich przeprowadzono testy, aby wybrać rozwiązanie najczęściej najlepiej się sprawujące. Tak prezentują się algorytmy:

Listing 1: swap()

```
1  def swap_sciezka(zaburzona_sciezka,a,b):
2      zaburzona_sciezka[a], zaburzona_sciezka[b] = zaburzona_sciezka[b],
           zaburzona_sciezka[a]
3      return zaburzona_sciezka
```

Listing 2: insert()

```
1  def insert_sciezka(zaburzona_sciezka,a,b,n):
2      while True:
3          if b+1==a:
4                  a,b = zaburzonePozycje(n)
5          else:
6              break
7      zaburzona_sciezka = np.insert(zaburzona_sciezka, a, zaburzona_sciezka[b
           ])
8      if a<b:
9          zaburzona_sciezka = np.hstack((zaburzona_sciezka[:b+1],
               zaburzona_sciezka[b+2:]))
10     else:
11         zaburzona_sciezka = np.hstack((zaburzona_sciezka[:b],
               zaburzona_sciezka[b+1:]))
12
13     return zaburzona_sciezka
```

Listing 3: invert()

```
def invert_sciezka(zaburzona_sciezka,a,b):
    if b<a:
        a, b = b, a
    while a<b:
        zaburzona_sciezka[a], zaburzona_sciezka[b] = zaburzona_sciezka[b],
            zaburzona_sciezka[a]
        a += 1
        b -= 1

    return zaburzona_sciezka
```

Algorytm zwracające losowo dobrane a oraz b:

Listing 4: zaburzonePozycje(n)

```
def zaburzonePozycje(n):
    a,b = 0,0
    while a == b:
        a,b = np.random.randint(1, high=n-1, size=2)
    return a,b
```

Algorytm obliczający długość ścieżki countPath(path, arr, rozmiar), w porównaniu do poprzedniego sprawozdania, nie uległ zmianom.

Aby algorytm działał skutecznie, należało wygenerować początkową ścieżkę. W przyjętym rozwiązaniu, początkowa ścieżka była ścieżka, o najkrótszej długości spośród n losowo wygenerowanych ścieżek:

Listing 5: inicjalizacja_poczatkowych_wartosci()

```
def inicjalizacja_poczatkowych_wartosci(arr,n):
    liczba_iteracji = n
    najkrotsza_sciezka = np.random.permutation(range(1,n))
    najlepszy_wynik = countPath(najkrotsza_sciezka, arr, n)
    for i in range(liczba_iteracji):
        losowa_sciezka = np.random.permutation(range(1,n))
        wynik = countPath(losowa_sciezka, arr, n)
        if wynik < najlepszy_wynik:
            najlepszy_wynik = wynik
            najkrotsza_sciezka = losowa_sciezka

    return najkrotsza_sciezka, najlepszy_wynik
```

2.1.1 Algorytm i jego przebieg

Algorytm wykonywał się dopóki jego obecna temperatura jest większa od temperatury końcowej, lub dopóki jego czas trwania nie przekracza 7s. Wartość 7s została dobrana w taki sposób, aby czas badania wszystkich wariantów algorytmu nie przekroczył 8h (testowanych było 5 rozmiarów, po 3 różne ilości iteracji, po 3 różne temperatury początkowe, po 3 różne rodzaje zmniejszania temperatury, po 3 różne permutacje losowych rozwiązań, po 10 powtórzeń każde, co łącznie daje 4050 wykonań algorytmu, co przy pesymistycznym czasie trwanie algorytmu, tj. 7s, daje prawie 8 godzin).
Algorytm na początku inicjuje wartości początkowe, ustawia wartości stałe (tj, max_czas oraz temperatura_koncowa),

zależnie od testowanego nr schematu wybiera ten schemat oraz ustawia wejściową ścieżkę, temperaturę oraz aktualny czas trwania algorytmu (2-18). Następnie rozpoczyna się główna pętla, która trwa dopóki niespełnione są żadne z warunków końcowych (20).
W pętli głównej zawarta jest kolejna pętla, która wykonuje się aktualnej liczbie iteracji razy (21). W pętli tej wybierane są miejsca w aktualnej ścieżce, które będziemy zaburzać, a następnie sposób zaburzenia tych miejsc (22-31). Następnie liczona jest długość zaburzonej ścieżki oraz jej różnica z długością poprzedniej ścieżki. Jeżeli długość zaburzonej ścieżki jest krótsza od długości poprzedniej ścieżki, to zapamiętujemy ją, jako nową ścieżkę, jeżeli jednak nie jest krótsza, to sprawdzamy, czy losowa liczba z przedziału 0-1 będzie mniejsza od $e^{(-roznica/temperatura)}$. Pozwala to przyjąć gorsze rozwiązanie, nie zatrzymując się w lokalnym minimum. Widać, że im mniejsza jest aktualna temperatura, tym wyrażenie będzie coraz mniejsze, tym samym, rzadziej będzie akceptowało gorsze rozwiązanie (33-40). Następnie sprawdzamy czy aktualne rozwiązanie jest lepsze niż aktualnie najlepsze rozwiązanie (41-42).
Po wyjściu z pętli, zależnie od przyjętego schematu, zmniejszamy temperaturę (46-51) oraz sprawdzamy aktualny czas wykonywania się algorytmu (53).

Listing 6: Algorytm Simulated Annealing

```
1  def obliczenieCzasuWyniku(arr, n, schemat, permutacja, liczba_iteracji,
       temperatura_poczatkowa):
2      start = datetime.datetime.now()
3      poczatkowa_sciezka, najkrotsza_dlugosc =
           inicjalizacja_poczatkowych_wartosci(arr,n)
4      i = 0
5      max_czas = 7.0
6      temperatura_koncowa = 0.0001
7
8      if schemat == 0:
9          lambda_liniowa = temperatura_poczatkowa/(n*10)
10     if schemat == 1:
11         lambda_geometryczna = 0.75
12     if schemat == 2:
13         lambda_logarytmiczna = 2.0
14
15     sciezka = poczatkowa_sciezka
16     temperatura = temperatura_poczatkowa
17     poprzednia_dlugosc = najkrotsza_dlugosc
18     duration = datetime.datetime.now() - start
19
20     while duration.total_seconds() < max_czas and temperatura >=
           temperatura_koncowa:
21         for m in range(liczba_iteracji):
22             a,b = zaburzonePozycje(n)
23             zaburzona_sciezka = sciezka.copy()
24             if permutacja == 0:
25                 zaburzona_sciezka = swap_sciezka(zaburzona_sciezka,a,b)
26
27             elif permutacja == 1:
28                 zaburzona_sciezka = insert_sciezka(zaburzona_sciezka,a,b,n)
29
30             else:
31                 zaburzona_sciezka = invert_sciezka(zaburzona_sciezka,a,b)
32
```

```
33              dlugosc = countPath(zaburzona_sciezka, arr, n)
34              roznica = dlugosc - poprzednia_dlugosc
35              if roznica <= 0:
36                  sciezka = zaburzona_sciezka
37                  poprzednia_dlugosc = dlugosc
38              elif np.random.rand() < math.exp(-roznica/temperatura):
39                  sciezka = zaburzona_sciezka
40                  poprzednia_dlugosc = dlugosc
41              if dlugosc < najkrotsza_dlugosc:
42                  najkrotsza_dlugosc = dlugosc
43
44          i += 1
45
46          if schemat == 1:
47              temperatura = lambda_geometryczna*temperatura #schemat
                    geometryczny
48          elif schemat == 2:
49              temperatura = temperatura/(1 + lambda_logarytmiczna*temperatura
                    ) #schemat logarytmiczny
50          else:
51              temperatura = temperatura - lambda_liniowa #schemat liniowy
52
53          duration = datetime.datetime.now() - start
54
55      return najkrotsza_dlugosc
```

2.2 Tabu Search

Algorytm Tabu Search (Poszukiwanie z zabronieniami) to algorytm metaheurystyczny naśladujący proces poszukiwania wykonywany przez człowieka. Algorytm wybiera lokalnie najlepsze rozwiązanie, unika rozwiązań, które w określonej długości przeszłości zostały odwiedzone, z tym, że i dla tych zdarza robić się wyjątki. Odwiedzone rozwiązania zapisuje na liście tabu, na której aktualnie może znajdować się pewna ilość ostatnio odwiedzonych rozwiązań. Jest to lista typu LIFO. Jeżeli rozwiązanie znajduje się na liście tabu, a daje obiecujące rozwiązanie, to wywoływana jest funkcja aspiracji, która sprawdza, czy rozwiązanie może zostać zaakceptowane (w badany algorytmie sprawdzono 2 funkcje aspiracji: porównanie do obecnie najlepszego wyniku oraz porównanie do poprzedniego wyniku). Dodatkowo, aby usprawnić algorytm, przy każdym pełnym poszukiwaniu lokalnym, tworzona jest lista master, która zapamiętuje obecnie 5 najlepszych rozwiązań. Pozwala to zwiększyć ilość operacji nawet 5 razy, ponieważ, dla dużych grafów, zmiana dokonana dla najlepszego rozwiązania, rzadko kiedy przedawni kolejne bardzo dobre rozwiązanie.

Rozwiązania losowe, podobnie jak w algorytmie SA, generowane są przez funkcje swap, insert oraz invert. Ponieważ tylko jedna z tych funkcji różni się od poprzednich, tylko ona zostanie tutaj wypisana:

Listing 7: insert(a b)

```
1  def insert_sciezka(zaburzona_sciezka,a,b,n):
2      if np.random.rand() > 0.5:
3          a, b = b, a
4      while True:
5          if b+1==a:
6                  a,b = zaburzonePozycje(n)
```

```
7           else:
8               break
9
10      zaburzona_sciezka = np.insert(zaburzona_sciezka, a, zaburzona_sciezka[b
            ])
11      if a<b:
12          zaburzona_sciezka = np.hstack((zaburzona_sciezka[:b+1],
                zaburzona_sciezka[b+2:]))
13      else:
14          zaburzona_sciezka = np.hstack((zaburzona_sciezka[:b],
                zaburzona_sciezka[b+1:]))
15
16      return zaburzona_sciezka, a, b
```

Ponieważ w tym algorytmie, a oraz b są generowane przez iterator, to a<b zawsze, co w swap i invert nie ma znaczenia, ma znaczenie w insert, dlatego na początku algorytmu, wprowadzono szansę 50% na zamianę wartości a z b.
Algorytm inicjujący początkowe wartości oraz zwracający zaburzone pozycje, nie różnią się od algorytmów w SA.

W algorytmie zaimplementowano następujące funkcje pomocnicze:

- Sprawdzanie czy ruch znajduje się na liście tabu:

Listing 8: sprawdzenie_listy_tabu(lista_tabu iteratool zaburzona_sciezka)

```
1 def sprawdzenie_listy_tabu(lista_tabu, iteratool, zaburzona_sciezka):
2     szukany_tuple = ()
3     a=zaburzona_sciezka[iteratool[1]]
4     b=zaburzona_sciezka[iteratool[0]]
5     szukany_tuple = tuple((a,b))
6     return szukany_tuple in lista_tabu
```

Funkcja przeszukuje listę tabu w celu sprawdzenia, czy podany iterator znajduje się na niej.

- Sprawdzanie pierwszego kryterium aspiracji:

Listing 9: sprawdzenie_kryterium_aspiracji1(najkrotsza_dlugosc dlugosc_zaburzonej_sciezki)

```
1 def sprawdzenie_kryterium_aspiracji1(najkrotsza_dlugosc,
      dlugosc_zaburzonej_sciezki):
2     return dlugosc_zaburzonej_sciezki < najkrotsza_dlugosc
```

Funkcja sprawdza, czy długość zaburzonej ścieżki, jest krótsza od najkrótszej dotychczasowo uzyskanej długości.

- Sprawdzanie drugiego kryterium aspiracji:

Listing 10: sprawdzenie_kryterium_aspiracji2(dlugosc dlugosc_zaburzonej_sciezki)

```
1 def sprawdzenie_kryterium_aspiracji2(dlugosc,
      dlugosc_zaburzonej_sciezki):
2     return dlugosc_zaburzonej_sciezki < dlugosc
```

Funkcja sprawdza, czy długość zaburzonej ścieżki, jest krótsza od ostatniej długości.

- Sortowanie słownika wg wartości:

Listing 11: sortuj_wg_wartosci(dictt)

```
1 def sortuj_wg_wartosci(dictt):
2     lst = sorted(dictt.items(), key=lambda kv: kv[1])
3     res_dct = {}
4     for i in range(len(lst)):
5         res_dct[lst[i][0]] = lst[i][1]
6     return res_dct
```

Funkcja sortuje słownik wg wartości. Funkcja przydatna aby wydobyć z sąsiadów 5 najlepszych rozwiązań.

2.2.1 Algorytm i jego przebieg

Na początku algorytmu, podobnie jak w SA, generowane są początkowe wartości oraz tworzone pusta lista tabu, pusty słownik master oraz wartość ostatniego elementu ze słownika master (w celu weryfikacji sensu sprawdzania pozostałych sąsiadów ze słownika master) (2-12). Następnie rozpoczyna się główna pętla, której warunkiem stopu jest przekroczenie 7s trwania algorytmu (14).
W pętli generujemy początkową długość (15) oraz jeżeli:

- nie mamy elementów w słowniku master to:
  dla wszystkich sąsiadów bieżącego rozwiązania, utworzonych wg jednego z trzech schematów, liczymy ich długość, sortujemy wg tej długości i tworzymy słownik master, przechowujący rozmiar_slownika_master elementów (19-45),
- w przeciwnym razie:
  zliczamy nowe odległości dla permutacji zawartych w słowniku master i sortujemy je (47-64).

Następnie wywoływana jest pętla, która domyślnie wykonuje się dla każdego elementu w słowniku master (66):

- zabierana jest pierwsza z góry droga,
- sprawdzane jest, czy jej długość, jest większa od wartości ostatniego elementu, wygenerowania podczas tworzenia słownika master i jeżeli nie, to słownik jest zerowany, a pętla przerywana (69-71),
- sprawdzenie, czy przejście znajduje się na liście tabu i jeżeli się znajduje, to zależnie od wybrane sposobu sprawdzania kryterium aspiracji, sprawdzana jest jego aspiracja (73-80),
- jeżeli przejścia nie ma na liście tabu lub jest, ale spełnia kryterium aspiracji, to zapisywana jest jego długość i ścieżka, a dane przejście zostaje zapisane jako przejscie_zakazane (82-86).

Po skończeniu pętli, sprawdzane jest, czy dane rozwiązanie jest najlepszym dotychczas uzyskanym rozwiązaniem (88-90), przejścia w liście tabu są weryfikowane (92-96), a następnie, jeżeli w danej iteracji uzyskaliśmy nowe rozwiązanie, to przejście jest dodawane do listy tabu (98-102).

Listing 12: Algorytm Tabu Search

```
1 def obliczenieCzasuWyniku(arr, n, permutacja, kryterium_aspiracji,
      rozmiar_slownika_master, dlugosc_listy_tabu):
2     start = datetime.datetime.now()
3     poczatkowa_sciezka, najkrotsza_dlugosc =
          inicjalizacja_poczatkowych_wartosci(arr,n)
4     i = 0
```

```
5       max_czas = 7.0
6       sciezka = poczatkowa_sciezka
7       najlepsza_sciezka = sciezka
8       lista_tabu = {}
9       slownik_master = {}
10      master_wartosc_ostatniego=0
11
12      duration = datetime.datetime.now() - start
13
14      while duration.total_seconds() < max_czas:
15          dlugosc = countPath(sciezka, arr, n,20)
16          przejscie_zakazane = ()
17          slownik_sasiadow = {}
18
19          if len(slownik_master) == 0:
20              for iteratool in itertools.combinations(range(0, n-1), 2):
21                  a = iteratool[0] # a<b zawsze
22                  b = iteratool[1]
23                  zaburzona_sciezka = sciezka.copy()
24
25                  if permutacja == 0:
26                      zaburzona_sciezka = swap_sciezka(zaburzona_sciezka,a,b)
27
28                  elif permutacja == 1:
29                      zaburzona_sciezka,a,b = insert_sciezka(
                            zaburzona_sciezka,a,b,n)
30
31                  else:
32                      zaburzona_sciezka = invert_sciezka(zaburzona_sciezka,a,
                            b)
33
34                  dlugosc_zaburzonej_sciezki = countPath(zaburzona_sciezka,
                        arr, n,permutacja)
35                  slownik_sasiadow[iteratool] = dlugosc_zaburzonej_sciezki
36
37              slownik_sasiadow = sortuj_wg_wartosci(slownik_sasiadow)
38              master_wartosc_ostatniego = 0
39              tmp = 0
40              for iteratool, dlugosc_zaburzonej_sciezki in slownik_sasiadow.
                    items():
41                  slownik_master[iteratool] = dlugosc_zaburzonej_sciezki
42                  tmp +=1
43                  master_wartosc_ostatniego = dlugosc_zaburzonej_sciezki
44                  if tmp == rozmiar_slownika_master:
45                      break
46
47          else:
48              for iteratool in slownik_master.keys():
49                  a = iteratool[0] # a<b zawsze
50                  b = iteratool[1]
```

```
51                  zaburzona_sciezka = sciezka.copy()
52
53                  if permutacja == 0:
54                      zaburzona_sciezka = swap_sciezka(zaburzona_sciezka,a,b)
55                  elif permutacja == 1:
56                      zaburzona_sciezka,a,b = insert_sciezka(
                            zaburzona_sciezka,a,b,n)
57                  else:
58                      zaburzona_sciezka = invert_sciezka(zaburzona_sciezka,a,
                            b)
59
60                  dlugosc_zaburzonej_sciezki = countPath(zaburzona_sciezka,
                        arr, n, 132)
61                  slownik_sasiadow[iteratool] = dlugosc_zaburzonej_sciezki
62
63              slownik_sasiadow = sortuj_wg_wartosci(slownik_sasiadow)
64              slownik_master = slownik_sasiadow
65
66          for iteratool, dlugosc_zaburzonej_sciezki in list(slownik_master.
                items()):
67              slownik_master.pop(iteratool)
68
69              if dlugosc_zaburzonej_sciezki > master_wartosc_ostatniego:
70                  slownik_master = {}
71                  break
72
73              czy_przejscie_na_liscie_tabu = sprawdzenie_listy_tabu(
                    lista_tabu, iteratool, zaburzona_sciezka)
74              czy_spelnia_kryterium_aspiracji = 0
75
76              if czy_przejscie_na_liscie_tabu:
77                  if kryterium_aspiracji == 1:
78                      czy_spelnia_kryterium_aspiracji =
                            sprawdzenie_kryterium_aspiracji1(najkrotsza_dlugosc,
                             dlugosc_zaburzonej_sciezki)
79                  else:
80                      czy_spelnia_kryterium_aspiracji =
                            sprawdzenie_kryterium_aspiracji2(dlugosc,
                            dlugosc_zaburzonej_sciezki)
81
82              if ~czy_przejscie_na_liscie_tabu or
                    czy_spelnia_kryterium_aspiracji:
83                  dlugosc = dlugosc_zaburzonej_sciezki
84                  sciezka = zaburzona_sciezka
85                  przejscie_zakazane = iteratool
86                  break
87
88          if dlugosc < najkrotsza_dlugosc:
89              najkrotsza_dlugosc = dlugosc
90              najlepsza_sciezka = sciezka
```

```
91
92          for klucz in list(lista_tabu):
93              if lista_tabu[klucz] == 1:
94                  lista_tabu.pop(klucz)
95              else:
96                  lista_tabu[klucz] -= 1
97
98          if len(przejscie_zakazane):
99              a = sciezka[przejscie_zakazane[1]]
100             b = sciezka[przejscie_zakazane[0]]
101             przejscie_zakazane = tuple((a,b))
102             lista_tabu[przejscie_zakazane] = dlugosc_listy_tabu
103
104         i += 1
105         duration = datetime.datetime.now() - start
106
107     return najkrotsza_dlugosc
```

3 Eksperymenty obliczeniowe

Obliczenia, tak samo jak poprzednio, zastały wykonane na laptopie z procesorem i5-7300HQ, kartą graficzną NVIDIA GeForce GTX 1050, 16GB RAM i DYSK SSD oraz zostały napisane w języku Python.

Dla algorytmu Simulated Annealing zebrano 405 danych, a dla algorytmu Tabu Search 270. Z powodu tak dużej liczby danych, wszystkie dane nie zostaną zaprezentowane, a jedynie to, co udało się na ich podstawie wywnioskować. Porównywane będą jakości wyników, które obliczane były poprzez podzielenie optymalnego rozwiązania, przez uzyskany wynik. Ponieważ uzyskany wynik, nie mógłby być mniejszy niż optymalny, wyniki mieszczą się w zakresie (0.0 - 1.0], gdzie 1.0 oznacza uzyskanie optymalnego wyniku, 0.5 uzyskanie dwukrotnie gorszego wyniku, itp.

3.1 Wnioski dla algorytmu Simulated Annealing

Porównanie jakości wyników wg następujących parametrów:

Tablica 1: Liczba iteracji

Liczba iteracji	min	mean	max
$Rozmiar * 5$	0.177	0.825	1.0
$Rozmiar^{3/2}$	0.642	0.891	1.0
$Rozmiar^2$	0.642	0.894	1.0

Tablica 2: Temperatura początkowa

Temperatura początkowa	min	mean	max
$Rozmiar * 20$	0.177	0.829	1.0
$Rozmiar^{3/2}$	0.631	0.890	1.0
$Rozmiar^2$	0.631	0.891	1.0

Tablica 3: Rodzaj schematu

Rodzaj schematu	min	mean	max
liniowy	0.177	0.848	1.0
geometryczny	0.239	0.873	1.0
logarytmiczny	0.631	0.888	1.0

Tablica 4: Rodzaj permutacji

Rodzaj permutacji	min	mean	max
swap	0.177	0.867	1.0
insert	0.177	0.870	1.0
invert	0.177	0.872	1.0

Z powyższych danych wynika, że dla:

- liczby iteracji:
  najlepiej wypadła ilość równa kwadratowi rozmiaru instancji, ale nie odbiega ona znacznie od ilości równej rozmiaru instancji podniesionej do potęgi 3/2, różnią się jedynie dla średnich, o trzy tysięczne. Rozmiar uzależniony od stałej liczby sprawdził się najgorzej.

- temperatury początkowej:
  najlepiej wypadła ilość równa kwadratowi rozmiaru instancji, ale nie odbiega ona znacznie od ilości równej rozmiaru instancji podniesionej do potęgi 3/2, różnią się jedynie dla średnich, o jedną tysięczną. Ponownie, gdy uzależniono rozmiar od stałej, otrzymano gorsze wyniki.

- rodzaju schematu:
  najlepiej wypadł schemat logarytmiczny, co mocno widać, patrząc na minimalny uzyskany wynik.

- rodzaju permutacji:
  najlepiej wypadła funkcja invert, jednak w porównaniu do swap i insert, nie różni się znacznie.

Najlepsze uzyskane jakości wyniku, dla poszczególnych rozmiarów:

Tablica 5: Najlepsze uzyskane wyniki

rozmiar	jakość wyniku	wynik	optymalny wynik
21	1.000	2707	2707
48	0.967	14918	14422
71	0.904	2156	1950
120	0.959	7242	6942
323	0.642	2065	1326

Jak widać, algorytmy dla niedużych wielkości instancji dają rozwiązania bliskie optymalnym, a nawet optymalne. Dla większych instancji otrzymany wynik różni się, jednak nadal nie jest 2x większy od optymalnego i dla różnych potrzeb mógłby zostać zaakceptowany.

Kombinacje parametrów, które dla wszystkich badanych instancji dały wyniki, równe najlepszym uzyskanym wynikom:

Tablica 6: Najlepsze kombinacje

liczba iteracji	temperatura początkowa	rodzaj schematu	rodzaj permutacji
$rozmiar^2$	$rozmiar * 20$	geometryczny	invert
$rozmiar^2$	$rozmiar * 20$	logarytmiczny	swap
$rozmiar^2$	$rozmiar * 20$	logarytmiczny	insert
$rozmiar^2$	$rozmiar * 20$	logarytmiczny	invert
$rozmiar^2$	$rozmiar^{3/2}$	liniowy	swap
$rozmiar^2$	$rozmiar^{3/2}$	liniowy	insert
$rozmiar^2$	$rozmiar^{3/2}$	liniowy	invert
$rozmiar^2$	$rozmiar^{3/2}$	geometryczny	swap
$rozmiar^2$	$rozmiar^{3/2}$	geometryczny	insert
$rozmiar^2$	$rozmiar^{3/2}$	geometryczny	invert
$rozmiar^2$	$rozmiar^{3/2}$	logarytmiczny	swap
$rozmiar^2$	$rozmiar^{3/2}$	logarytmiczny	insert
$rozmiar^2$	$rozmiar^{3/2}$	logarytmiczny	invert
$rozmiar^2$	$rozmiar^2$	liniowy	swap
$rozmiar^2$	$rozmiar^2$	liniowy	insert
$rozmiar^2$	$rozmiar^2$	liniowy	invert
$rozmiar^2$	$rozmiar^2$	geometryczny	swap
$rozmiar^2$	$rozmiar^2$	geometryczny	insert
$rozmiar^2$	$rozmiar^2$	geometryczny	invert
$rozmiar^2$	$rozmiar^2$	logarytmiczny	swap
$rozmiar^2$	$rozmiar^2$	logarytmiczny	insert
$rozmiar^2$	$rozmiar^2$	logarytmiczny	invert

Jak widać w zestawieniu, największe znaczenie miał wybór liczby iteracji, tylko dla liczby iteracji $rozmiar^2$ kombinacje dały najlepsze wyniki. W przypadku uzależnienia temperatury początkowej od rozmiaru pomnożonego przez stałą, najlepszy wynik uzyskano, gdy schemat był logarytmiczny a permutacja dowolna, lub schemat był geometryczny, a permutacja invert. W przypadku uzależnienia temperatury początkowej wyłącznie od rozmiaru podniesionego do potęgi 3/2 lub 2, wybór schematu lub permutacji nie wpływał na jakość wyniku.

3.2 Wnioski dla algorytmu Tabu Search

Porównanie jakości wyników wg następujących parametrów:

Tablica 7: Rozmiar słownika master

Rozmiar słownika master	min	mean	max
$Rozmiar^{1/2}$	0.213	0.520	0.949
$Rozmiar$	0.213	0.541	0.949
$Rozmiar^2$	0.213	0.541	0.949

Tablica 8: Długość listy tabu

Długość listy tabu	min	mean	max
7	0.213	0.526	0.949
10	0.213	0.536	0.949
$Rozmiar$	0.213	0.539	0.949

Tablica 9: Kryterium aspiracji

Kryterium aspiracji	min	mean	max
1 - dlugosc < najkrotsza dlugosc	0.213	0.530	0.949
2 - dlugosc < poprzednia dlugosc	0.213	0.538	0.949

Tablica 10: Rodzaj permutacji

Rodzaj permutacji	min	mean	max
swap	0.213	0.532	0.949
insert	0.213	0.533	0.949
invert	0.213	0.537	0.949

Z powyższych danych wynika, że dla:

- rozmiaru słownika master:
  najlepiej wypadły ilości równe rozmiarowi oraz rozmiarowi do kwadratu danej instancji.
- długości listy tabu:
  najlepiej wypadła ilość równa rozmiarowi instancji. Trochę gorzej (o trzy tysięczne) wypadła długość równa 10. Długość równa 7 wypadła najgorzej, jednak jedynie o dziesięć tysięcznych gorzej od długości równej 10.
- kryterium aspiracji:
  najlepiej wypadło kryterium porównujące długość do poprzedniej długości. Może to wynikać z większej otwartości algorytmu na gorsze rozwiązania.
- rodzaju permutacji:
  najlepiej wypadła funkcja invert, jednak w porównaniu do swap i insert, nie różni się znacznie.

Najlepsze uzyskane jakości wyniku, dla poszczególnych rozmiarów:

Tablica 11: Najlepsze uzyskane wyniki

rozmiar	jakość wyniku	wynik	optymalny wynik
21	0.949	2851	2707
48	0.824	17494	14422
71	0.475	4102	1950
120	0.213	32626	6942
323	0.243	5467	1326

Jak widać, algorytm Tabu Serach poradził sobie gorzej. Nawet dla małej instancji nie dał optymalnego wyniku. Już od wielkości 71, wynik jaki daje, jest ponad 2x gorszy od optymalnego, dla rozmiaru 323, ponad 4x gorszy, a dla rozmiaru 120 prawie 5x gorszy.

Kombinacje parametrów, które dla wszystkich badanych instancji dały wyniki, równe najlepszym uzyskanym wynikom:

Tablica 12: Najlepsze kombinacje

rozmiar słownika master	długość listy tabu	kryterium aspiracji	rodzaj permutacji
$\sqrt{rozmiar}$	rozmiar	2	swap
$\sqrt{rozmiar}$	rozmiar	2	insert
$\sqrt{rozmiar}$	rozmiar	2	invert
rozmiar	7	1	swap
rozmiar	7	1	insert
rozmiar	7	1	invert
rozmiar	7	2	swap
rozmiar	7	2	insert
rozmiar	7	2	invert
rozmiar	10	1	swap
rozmiar	10	1	insert
rozmiar	10	1	invert
rozmiar	10	2	swap
rozmiar	10	2	insert
rozmiar	10	2	invert
rozmiar	rozmiar	1	swap
rozmiar	rozmiar	1	insert
rozmiar	rozmiar	1	invert
rozmiar	rozmiar	2	swap
rozmiar	rozmiar	2	insert
rozmiar	rozmiar	2	invert
$rozmiar^2$	7	1	swap
$rozmiar^2$	7	1	insert
$rozmiar^2$	7	1	invert
$rozmiar^2$	7	2	swap
$rozmiar^2$	7	2	insert
$rozmiar^2$	7	2	invert
$rozmiar^2$	10	1	swap
$rozmiar^2$	10	1	insert
$rozmiar^2$	10	1	invert
$rozmiar^2$	10	2	swap
$rozmiar^2$	10	2	insert
$rozmiar^2$	10	2	invert
$rozmiar^2$	rozmiar	1	swap
$rozmiar^2$	rozmiar	1	insert
$rozmiar^2$	rozmiar	1	invert
$rozmiar^2$	rozmiar	2	swap
$rozmiar^2$	rozmiar	2	insert
$rozmiar^2$	rozmiar	2	invert

Jak widać w zestawieniu, większość kombinacji znalazła się w nim. Jeżeli rozmiar słownika master był równy rozmiarowi lub kwadratowi rozmiaru instancji, pozostałe parametry nie wpływały na rozwiązanie. W przypadku gdy rozmiar słownika master wynosił pierwiastek rozmiaru instancji, to najlepsze wyniki uzyskano tylko wtedy, gdy długość listy tabu była równa rozmiarowi instancji, a wybrane kryterium aspiracji porównywało wynik, to poprzedniego wyniku, a nie najlepszego.

3.3 Porównanie algorytmów

Oto wyniki prezentujące jak ma się najlepsza jakość wyniku algorytmu Tabu Search, do najlepszej jakości wyniku algorytmu Simulated Annealing:

Tablica 13: Porównanie Tabu Search do Simulated Annealing

rozmiar	TS/SA
21	0.949
48	0.853
71	0.526
120	0.222
323	0.378

4 Wnioski

Algorytmy Simulated Annealing oraz Tabu Search wykonywały się w tym samym czasie (poza momentami, gdy SA zostało przerwane z powodu zbyt niskiej temperatury), a mimo tego, przy większych rozmiarach instancji, algorytm SA okazał się nawet kilkukrotnie lepszy od algorytmu TS. Jednak analizując wyniki otrzymane przy pomocy algorytmu TS, zwłaszcza te zawarte w Tablica 12 oraz w Tablica 11. widać, że algorytm daje słabe wynika, a dodatkowo, dla 39 z 54 badanych kombinacji daje te same wyniki, dla wszystkich rozmiarów, co może świadczyć o błędnej implementacji algorytmu powodującej powtarzanie przez te kombinacje podobnych błędów i utykaniu w tym samym lokalnym minimum.